Auto-Active Proof of Red-Black Trees in SPARK*

Claire Dross and Yannick Moy

AdaCore, F-75009 Paris

Abstract Formal program verification can guarantee that a program is free from broad classes of errors (like reads of uninitialized data and run-time errors) and that it complies with its specification. Tools such as SPARK make it cost effective to target the former in an industrial context, but the latter is much less common in industry, owing to the cost of specifying the behavior of programs and even more the cost of achieving proof of such specifications. We have chosen in SPARK to rely on the techniques of auto-active verification for providing cost effective formal verification of functional properties. These techniques consist in providing annotations in the source code that will be used by automatic provers to complete the proof. To demonstrate the potential of this approach, we have chosen to formally specify a library of red-black trees in SPARK, and to prove its functionality using auto-active verification. To the best of our knowledge, this is the most complex use of auto-active verification so far.

1 Introduction

Formal program verification allows programmers to guarantee that the programs they write have some desired properties. These properties may simply be that the program does not crash or behave erratically, or more complex critical properties related to safety or security. Being able to guarantee such properties will be essential for high assurance software as requirements are increasingly complex and security attacks more pervasive.

SPARK is a subset of the Ada programming language targeted at safety- and security-critical applications. GNATprove is a tool that analyzes SPARK code and can prove absence of run-time errors and user-specified properties expressed as contracts. GNATprove is based on modular deductive verification of programs, analyzing each function in isolation based on its contract and the contracts of the functions it calls. The main benefit of this approach is that it allows using very precise semantics of programming constructs and powerful automatic provers. The main drawback is that top-level specifications are not sufficient. Programmers need to provide many intermediate specifications in the form of additional contracts, loop invariants and assertions.

* Work partly supported by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, http://www.spark-2014.org/proofinuse) and project VECOLIB (ANR-14-CE28-0018) of the French national research organization.

Providing the right intermediate specifications is a difficult art, but progress has been achieved in recent years through a method known as auto-active verification. Various languages and tools now provide features for effective auto-active verification. SPARK is among these. In this paper, we explore the capabilities of auto-active verification for automatically proving complex algorithms. We have chosen to target red-black trees because they are well-known, commonly used in practice, and yet sufficiently complex that no implementation of imperative red-black trees has been formally verified using auto-active verification. Our implementation of red-black trees, with all the code for auto-active verification, is publicly available in the repository of SPARK. [1]

2 Preliminaries

2.1 SPARK 2014

SPARK is a subset of the Ada programming language targeted at safety- and security-critical applications. SPARK builds on the strengths of Ada for creating highly reliable and long-lived software. SPARK restrictions ensure that the behavior of a SPARK program is unambiguously defined, and simple enough that formal verification tools can perform an automatic diagnosis of conformance between a program specification and its implementation. The SPARK language and toolset for formal verification have been applied over many years to on-board aircraft systems, control systems, cryptographic systems, and rail systems [18].

In the versions of SPARK up to SPARK 2005, specifications are written as special annotations in comments. Since version SPARK 2014 [17], specifications are written as special Ada constructs attached to declarations. In particular, various contracts can be attached to subprograms: data flow contracts, information flow contracts, and functional contracts (preconditions and postconditions, introduced respectively by `Pre` and `Post`). An important difference between SPARK 2005 and SPARK 2014 is that functional contracts are executable in SPARK 2014, which greatly facilitates the combination of test and proof. The definition of the language subset is motivated by the simplicity and feasibility of formal analysis and the need for an unambiguous semantics. Tools are available that provide flow analysis and proof of SPARK programs.

Flow analysis checks correct access to data in the program: correct access to global variables (as specified in data and information flow contracts) and correct access to initialized data. Proof is used to demonstrate that the program is free from run-time errors such as arithmetic overflow, buffer overflow and division-by-zero, and that the functional contracts are correctly implemented. GNATprove is the tool implementing both flow analysis and proof of SPARK code.

[1] https://github.com/AdaCore/spark2014/tree/master/testsuite/gnatprove/tests/red_black_trees

2.2 Auto-active Verification

The term *auto-active verification* was coined in 2010 by researcher Rustan Leino [16] to characterise *tools where user input is supplied before VC generation [and] therefore lie between automatic and interactive verification* (hence the name auto-active). This is in contrast to fully automatic verifiers for which *the specification is fixed* and interactive verifiers for which *the user input is supplied after VC generation, which is the typical case when the reasoning engine is an interactive proof assistant.* Auto-active verification is at the center of the academic formal program verification toolsets Dafny [14], the Eiffel Verification Environment (EVE) [9], Why3 [8] as well as the industrial formal program verification toolsets Frama-C [2] and SPARK [3].

In all these toolsets, auto-active verification consists in a set of specification features at the level of the source language, and a set of tool capabilities to interact with users at the level of the source code. The specification features consist at least in constructs to specify function contracts (preconditions and postconditions) and data invariants, as well as specialized forms of assertions (loop invariants and loop variants, assumptions and assertions). All the toolsets mentioned above also support *ghost code*, a feature to instrument code for verification. Ghost functions are also called lemmas when their main purpose is to support the proof of a property that is later used at the point where the function is called. See [12] for a comparison of how ghost code differs between Why3, Frama-C and SPARK. Various tool capabilities facilitate user interaction at source level: fast running time that exploits multiprocessor architectures and minimizes rework between runs, the ability to trade running time for more verification power, feedback from the toolset when verification is unsuccessful (counterexamples in particular).

Auto-active verification in the above toolsets has been used to fully verify algorithms, libraries and even full applications: examples include a container library in Eiffel [19], distributed systems in Dafny [10], secure execution of apps in Dafny [11], binary heaps in Why3 [21], allocators in SPARK [5].

2.3 Red-Black Trees

Red-black trees are a kind of self-balancing binary search trees. Nodes in the tree are colored red or black, and balance is maintained by ensuring that two properties are preserved: (1) a red node can only have black children, and (2) every path from the root to a leaf has the same number of black nodes. The consequence of these two properties is that the path from the root to a leaf can be at most twice as long as the path from the root to another leaf.

Implementations of red-black trees are used in the Linux kernel (in C) and standard container libraries for various languages (C++ STL, Java.util, Ada). The insertion and deletion algorithms work by inserting or deleting the node

[2] http://frama-c.com/
[3] http://www.adacore.com/sparkpro/

as in a binary search tree, which may violate properties (1) and (2) above, and then restoring the balance by working their way up on the path from the root to the point of insertion or deletion. At every node on this path, the algorithms may *rotate* the subtree, which consists in a local rearrangement of nodes to restore properties (1) and (2). These algorithms are sufficiently complex that no implementation of imperative red-black trees has been formally verified in Dafny, Eiffel or Why3. See Section 5 for a list of the closest works, including some using auto-active verification. We are following the algorithm from Cormen et al. [4] for insertion in a red-black tree. We did not implement the deletion algorithm, which would be very similar to insertion. In the same way, we did not verify that every branch in a red-black tree contains the same number of black nodes.

3 Red-Black Trees in SPARK

3.1 Invariants and Models

Implementing red-black trees correctly from the pseudo-code algorithm in a textbook is straightforward, but understanding why the algorithm is correct is tricky, and thus the implementation is hard to verify formally. The main point of complexity is that it forces one to reason about different levels of properties all at once. Instead, we have divided the implementation into three distinct parts, each one concerned with one property level: binary trees, search trees and red-black trees. Binary trees maintain a tree structure on the underlying memory. Search trees build on binary trees by associating values to tree nodes and maintain the order of values in the tree. Red-black trees build on search trees and enforce balancing using the classical red-black tree coloring mechanism.

The property enforced at each level is expressed in a type invariant. In SPARK, the invariant may be temporarily violated inside the implementation of the functions that operate on the type, but are guaranteed to hold for external users of objects of that type. More precisely, functions that operate on a type can assume the invariant on entry and must restore it on exit (which leads to verification conditions in SPARK).

Binary trees: As explained in Section 3.2, binary trees are implemented as arrays, using the representation described in Figure 1. Each node contains a reference to its left and right children, if any, as well as a reference to its parent and a position, which may be Top for the root, Right or Left otherwise depending on the node position with respect to its parent. The invariant of binary trees states that values of these fields are consistent across the tree. For example, the left child of a node has position Left and the node as parent.

To reason about the tree structure at a higher level, we provide a model (an abstract representation) of binary trees which makes explicit the *access paths* from the root to every node in the tree. It associates a sequence of directions, namely Right or Left, with each node in the binary tree, corresponding to the path from the root to the node. As the underlying array also contains unused cells that do not correspond to tree nodes, an additional boolean encodes whether the

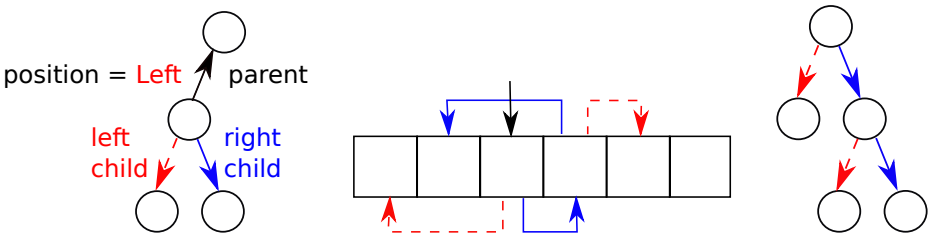

Fig. 1. (from left to right) Representation of nodes in binary trees. Example of a binary tree, for readability, parents and positions are not represented. A higher level view of the same binary tree.

node belongs to the tree. Figure 2 gives the model of the binary tree presented in Figure 1. In this example, all the nodes belong to the tree except the last one. The access paths written below each node can be used to reconstruct easily the high level view of the tree.

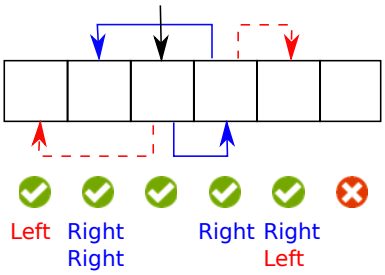

Fig. 2. Example of model of a binary tree.

Search trees: The invariant of search trees states that the value stored in each node of the tree is larger than all the values stored in the subtree rooted at its left child and smaller than all the values stored in the subtree rooted at its right child. It is given in Figure 3, together with an example of values that would fit the tree from Figure 1. To express this invariant, we use the model of the underlying binary tree. The value stored at node `J` belonging to the subtree rooted at node `I` (where path inclusion from the root is used to determine that `J` belongs to the subtree rooted at node `I`) is smaller (resp. greater) than the value stored at node `I` if `J` belongs to the subtree rooted at the left (resp. right) child of `I`.

Red-black trees: The invariant of red-black trees states that a red node can only have black children. It is given in Figure 4. An example of colors that would fit the tree from Figure 3 is also given in Figure 4. This corresponds to property (1) of red-black trees as presented in Section 2.3. Verifying property (2) would require implementing a new inductive model function over binary trees, like the one we defined for reachability. As it would be very similar to the work presented here, and would essentially double the effort, we did not attempt it.

```
(for all I in Index_Type ⇒
  (for all J in Index_Type ⇒
    (if Model (T) (I).Reachable
      and Model (T) (J).Reachable
      and Model (T) (I).Path < Model (T) (J).Path
     then (if Get (Model (T) (J).Path,
                   Length (Model (T) (I).Path) + 1) = Left
           then Values (J) < Values (I)
           else Values (J) > Values (I)))))
```

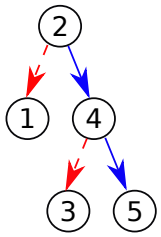

Fig. 3. Type invariant of search trees. For a search tree `T`, `Model (T)` returns the model of the underlying binary tree of `T`. For each index `I` in the underlying array, if `Model (T) (I).Reachable` is true then `I` is reachable in `T` and `Model (T) (I).Path` is the sequence of directions corresponding to the path from the root of `T` to `I`. $<$ stands for prefix order on paths.

```
(for all I in Index_Type ⇒
   (if Parent (T.Struct, I) = Empty
     or else T.Color (Parent (T.Struct, I)) = Red
    then T.Color (I) = Black))
```

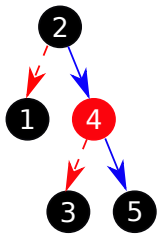

Fig. 4. Type invariant of red-black trees.

3.2 Implementation

Our implementation of red-black trees differs on two accounts from the straighforward implementation of the algorithm. First, as stated above, we used an array as the underlying memory for trees, instead of dynamically allocating nodes. This is to comply with a restriction of SPARK which does not allow pointers, but only references and addresses. The rationale for this restriction is that pointers make automatic proof very difficult due to possible aliasing. Hence trees are bounded by the size of the underlying array. As the algorithm for balancing red-black trees requires splitting and merging trees, we had the choice of either copying arrays for generating new trees, or sharing the same array between disjoint trees (coming from the splitting of a unique tree). For obvious efficiency reasons, we chose the latter. Hence we are defining a type `Forest` for possibly representing disjoint binary trees sharing the same underlying array.

The other distinguishing feature of our implementation is the layered design. Each module defining a type with an invariant also needs to provide functions for manipulating objects of the type while preserving their invariant. As an example, binary trees are not updated by direct assignments in the implementation of search trees, but using two new functions, `Extract` and `Plug`, which split and merge disjoint trees while preserving the forest invariant.

At the next layer, search trees are defined as records with two components: a binary tree along with an additional array of values. For search trees, we only need to consider forests that hold one tree identified through its root. Only intermediate values will hold true forests with multiple roots, while the tree is

being rotated. The module defining search trees provides basic set functions, namely inserting a value into the tree and testing a value for membership in the tree. It also provides balancing functions for the upper layer of red-black trees. They allow rotating nodes of a search tree to the left or to the right while preserving the order between values. An example of such a rotation is given in Figure 5. Defining these balancing functions inside the implementation of search trees rather than inside the implementation of red-black trees allows keeping all order-related concerns in the search tree layer. Indeed, balancing functions do not preserve balance, as they are to be called on unbalanced trees, but they do preserve order. Note that implementing the balancing functions at this level avoids the need for lifting low-level tree handling functions such as `Plug` and `Extract` at the next layer. All the functions defined on search trees are implemented using functions over binary trees.

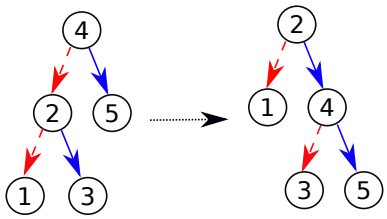

Fig. 5. Example of application of Right_Rotate.

Red-black trees are implemented in the same way as search trees by adding an array of colors to a search tree and using balancing functions to rebalance the tree after an insertion.

3.3 Specification

Functional specifications of the insertion and membership functions that operate on red-black trees consist in simple contracts (preconditions and postconditions) presented in Figure 6. These contracts use a model function `Values` that returns the set of values in the tree. `Mem` returns true if and only if the element is in the tree and `Insert` adds a new element in the tree.

```
function Values (T : Rbt) return Value_Set with
  Post => (if Size (T) = 0 then Is_Empty (Values'Result));

function Mem (T : Rbt; V : Natural) return Boolean with
  Post => Mem'Result = Mem (Values (T), V);

procedure Insert (T : in out Rbt; V : Natural) with
  Pre  => Size (T) < Max,
  Post => (if Mem (T'Old, V) then Values (T) = Values (T'Old)
           else Is_Add (Values (T'Old), V, Values (T)));
```

Fig. 6. Specification of red-black trees.

The most complex specifications have to do with the four properties to maintain over red-black trees:

1. A red-black tree is always a valid binary tree (we can navigate it from the root in the expected way).
2. There is no memory leak (if we have inserted fewer than `Max` elements, there is still room enough in the data structure to insert a new element).
3. The values stored in the tree are ordered (it is a valid search tree).
4. The tree stays balanced (we only verify this property partially, that is, that red nodes can only have black children).

As already discussed, each property is specified at the most appropriate layer. The first property is enforced at the level of binary trees. The invariant on binary trees (see Section 3.1) ensures that the fields of a node (Parent, Position, Left, and Right) are consistent. This is not enough to ensure that all the allocated nodes in the forest belong to well-formed binary trees though, as it does not rule out degenerate, root-less, cyclic structures that would arise from linking the root of a binary tree as the child of one of its leafs. Still, this is enough to ensure that red-black trees are always well formed, as red-black trees always have a root. Note that the fact that every node in the forest is part of a well formed binary tree is ensured at the level of binary trees by enforcing that such degenerate structures can never be created in the contracts of functions operating on binary trees.

The second property is enforced at the level of search trees. It is specified as a postcondition of every function operating on search trees. Figure 7 shows the part of the postcondition of `Right_Rotate` ensuring that it has not introduced any dangling node. It uses the function `Model` described in Section 3.1 to reason about node reachability.

```
procedure Right_Rotate (T : in out Search_Tree; I : Index_Type) with
  Post =>
    --  The size of the tree is preserved
    Size (T) = Size (T)'Old

    --  Nodes in the tree are preserved
    and (for all J in Index_Type =>
           Model (T) (J).Reachable = Model (T'Old) (J).Reachable);
```

Fig. 7. Postcondition of `Right_Rotate` dealing with absence of memory leaks.

The third and fourth properties are expressed in the type invariant of respectively search trees and red-black trees as explained in Section 3.1.

Apart from these top-level specifications, many more specifications are needed on subprograms at lower layers (binary trees and search trees) in order to be able to prove the properties at higher layers (respectively search trees and red-black trees). This is inherent to the modular style of verification supported by GNATprove. For example, as `Right_Rotate` on search trees calls `Plug` and `Extract` on binary trees, the contracts for these functions need to provide enough information to verify both the absence of memory leaks as stated in the

postcondition of `Right_Rotate` and the preservation of the order of values as stated in the invariant of search trees.

3.4 Proof Principles

Verifying our implementation of red-black trees has proved to be challenging, and above the purely automatic proving capabilities of GNATprove. There are several reasons for this:

- The imperative, pointer-based implementation of red-black trees makes it difficult to reason about disjointness of different trees/subtrees in the forest.
- Reasoning about reachability in the tree structure involves inductive proofs, which automatic provers are notoriously bad at.
- Reasoning about value ordering involves using transitivity relations, to deduce that ordering for two pairs of values (X, Y) and (Y, Z) can be extended to the pair (X, Z). This requires in general to find a suitable intermediate value Y, which usually eludes automatic provers.
- The size of the formulas to verify, number of verification conditions, and number of paths in the program are large enough to defy provers scalability.

To work around these limitations, we used auto-active verification techniques, which, as described in Section 2.2, can guide automatic provers without requiring a proof assistant. We explain some of these techniques in this section.

Intermediate lemmas: One of the classical techniques in manual proof consists in factoring some useful part of a proof in an intermediate lemma so that it can be verified independently and used as many times as necessary. In auto-active verification, this can be done by introducing a procedure with no output, which, when called, will cause the deductive engine to verify its precondition and assume its postcondition. In Figure 8, we show an intermediate lemma which can be used to verify that two trees of a single forest with different roots are disjoint. A caller of this function will have to verify that `T1` and `T2` are different valid roots in `F` and as a consequence we know that there can be no node reachable from both roots in `F`. Naturally, the lemma is not assumed, its actual proof is performed when verifying the procedure `Prove_Model_Distinct`.

```
procedure Prove_Model_Distinct (F : Forest; T1, T2 : Index_Type) with
--  Trees rooted at different indexes in the forest are disjoint.
  Pre  => T1 /= T2
    and then Valid_Root (F, T1)
    and then Valid_Root (F, T2),
  Post => (for all I in Index_Type =>
             (not Model (F, T1) (I).Reachable
                or not Model (F, T2) (I).Reachable));
```

Fig. 8. Intermediate lemma stating disjointness of trees in a forest.

Reasoning by induction: Though some automatic provers are able to discharge simple inductive proofs, inductive reasoning still requires manual interaction in most cases. In auto-active style, an inductive proof can be done using loop invariants. GNATprove splits the verification of a loop invariant in two parts. First, it verifies that the invariant holds in the first iteration of the loop and then that it holds in any following iteration knowing that it held in the previous one. This behavior is exactly what we want for a proof by induction. For example, Figure 9 demonstrates how the intermediate lemma presented in Figure 8 can be verified using a loop to perform an induction over the size of the path from the root `T1` to any node reachable from `T1` in `F`. The loop goes from 1 to the maximum size of any branch in the forest `F`. We have written the property we wanted to prove as a loop invariant. To verify this procedure, GNATprove will first check that the invariant holds in the first iteration of the loop, that is, that `T1` itself cannot be reached from `T2`. Then, it will proceed by induction to show that this holds for any node reachable from `T1` in `F`.

```
procedure Prove_Model_Distinct
   (F : Forest; T1, T2 : Index_Type) is
begin
   for N in Index_Type loop
      pragma Loop_Invariant
        (for all I in Index_Type ⇒
          (if Model (F, T1) (I).Reachable
             and Length (Model (F, T1) (I).Path) < N
           then not Model (F, T2) (I).Reachable));
   end loop;
end Prove_Model_Distinct;
```

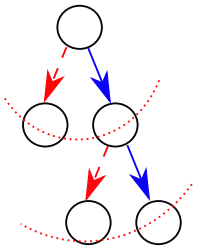

Fig. 9. Proof by induction over the path length from the root to a node in the tree.

Providing witnesses: When reasoning about value ordering, it is common to use transitivity. For example, when searching for a value in a search tree, we only compare the requested value with values stored along a single path in the tree, that is, the path where it was expected to be stored. All other values are ruled out by transitivity of the order relation: if value X is not found on this path, it cannot be equal to another value Z in the tree, as X and Z are on two opposite sides of the value Y at the root of the subtree containing both X and Z. Unfortunately, due to how they handle universal quantification, automatic provers used in GNATprove are usually unable to come up with the appropriate intermediate value to use in the transitivity relation. To achieve the proofs, we provided provers with the appropriate term whenever necessary. For example, function `Find_Root` in Figure 10 computes the first common ancestor of two nodes in a search tree.

3.5 Ghost Code

In this experiment, we made an extensive use of ghost code, that is, code meant only for verification, that has no effect on the program behavior. We used it

```
function Find_Root (F : Forest; R, I, J : Index_Type) return Index_Type with
  Post =>
    --  The node returned is in the tree
    Model (F, R) (Find_Root'Result).Reachable

    --  The node returned is on the path of I
    and Model (F, R) (Find_Root'Result).Path ≤ Model (F, R) (I).Path

    --  The node returned is on the path of J
    and Model (F, R) (Find_Root'Result).Path ≤ Model (F, R) (J).Path

    --  The common ancestor of I and J is either I, or J, or an ancestor
    --  node such that the paths of I and J diverge at this point.
    and (I = Find_Root'Result
           or else J = Find_Root'Result
           or else Get (Model (F, R) (I).Path,
                        Length (Model (F, R) (Find_Root'Result).Path) + 1)
                ≠ Get (Model (F, R) (J).Path,
                        Length (Model (F, R) (Find_Root'Result).Path) + 1));
```

Fig. 10. Function that computes a witness for transitivity applications.

for two different purposes. The first use of ghost code is for specifying complex properties about our algorithms, in particular through model functions. As ghost code can be executed in SPARK, these ghost model functions can be used to produce complex test oracles that can be exercised in the test campaign.

The second use of ghost code in our experiment is for auto-active verification. In particular, the procedures used to encode intermediate lemmas are ghost, as they have no effect. What is more, we strived to keep all verification-only code inside ghost procedures so that it can be removed by the compiler and will not slow down the execution of the program. It is all the more important since the code is very inefficient, involving multiple loops and model constructions. As functional behaviors are complex, coming up with contracts for these ghost procedures can be painful, and produce huge, hard to read specifications. To alleviate this problem, we can benefit from a feature of GNATprove which inlines local subprograms with no contracts, allowing the proof to go through with less annotation burden. In this way, we can choose, on a case-by-case basis, if it is worthwhile to turn a chunk of auto-active proof into an intermediate lemma with its own contract, allowing for a modular verification, or if we prefer to have the tool automatically inline the proof wherever we call the ghost procedure.

4 Development and Verification Data

All the execution times and verification times reported in this section were obtained on a Core i7 processor with 2,8 GHz and 16 GB RAM.

The code implementing the core algorithm for red-black trees, even when split in three modules for binary trees, search trees and red-black trees, is quite small, only 286 lines overall. But this code only accounts for 14% of the total lines of code, when taking into account contracts (22%) and more importantly ghost code (64%). Table 1 summarizes the logical lines of code as counted by the tool GNATmetric. It took roughly two weeks to develop all the code, contracts and ghost code to reach 100% automatic proof.

	code	contracts	ghost	total
binary trees	92 (10%)	250 (28%)	548 (62%)	890
search trees	127 (12%)	188 (17%)	780 (71%)	1095
red-black trees	67 (52%)	18 (14%)	45 (35%)	130
total	286 (14%)	456 (22%)	1373 (64%)	2115

Table 1. Number of lines of code for operational code, contracts and ghost code.

There are few simple top-level contracts for red-black trees (see Table 2). Many more contracts and assertions are needed for auto-active verification, in the form of subprogram contracts, type invariants, type default initial conditions, loop invariants and intermediate assertions which split the work between automatic provers and facilitate work of individual provers.

	on types	on subprograms	on loops	assertions	total
binary trees	10	155 (73)	42	12	219
search trees	2	138 (60)	20	68	228
red-black trees	2	4 (4)	8	10	24
total	14	297 (177)	70	90	471

Table 2. Number of conjuncts (and-ed subexpressions) in contracts on types, on subprograms, in loop invariants and in assertions. Numbers in parentheses correspond to conjuncts for contracts on externally visible subprograms.

Taking both tables into account, it is clear that verification of search trees was the most costly in terms of overall efforts, with a large part of ghost code (71%) and many intermediate assertions needed (68 conjuncts). Verification of red-black trees on the contrary was relatively straighforward, with less ghost code than operational code (35% compared to 52%) and few intermediate assertions needed (10 conjuncts). This matches well the cognitive effort required to understand the correction of search trees compared to red-black trees. Note that the verification of red-black trees would probably have needed roughtly the same effort as binary trees if the second propery of red-black trees had been considered. Overall, ghost code accounts for a majority (64%) of the code, which can be explained by the various uses of ghost code to support automatic proof as described in Section 3.4.

The automatic verification that the code (including ghost code) is free of run-time errors and that it respects its contracts takes less than 30 minutes, using 4 cores and two successive runs of GNATprove at proof levels 2 and 3. As automatic provers CVC4, Z3 and Alt-Ergo are called in sequence on unproved Verification Conditions (VCs), it is not surprising that CVC4 proves a majority of VCs (3763), while Z3 proves 103 VCs left unproved by CVC4 and Alt-Ergo proves the last 3 remaining VCs, for a total of 3869 VCs issued from 2414 source code checks (1185 run-time checks, 231 assertions and 998 functional contracts).

As the code has been fully proved to be free of run-time errors and that all contracts have been proved, it is safe to compile it with no run-time checks, and only the precondition on insertion in red-black trees activated (since this might

be violated by an external call). Disabling run-time checks is done through a compiler switch (-gnatp) and only enabling preconditions in red-black trees is done through a configuration pragma in the unit. Inserting one million integers in a red-black tree from 1 to 1 million leads to a violation of the balancing in 999,998 cases, which requires 999,963 left rotations and no right rotations. The running time for performing these 1 million insertions is 0.65 seconds without run-time checks, and 0.70 seconds with run-time checks (which are few due to the use of Ada range types for array indexes), or 0.65 microseconds (respectively 0.70 microseconds) per insertion.

Enabling all contracts and assertions at run-time is also possible during tests. Here, ghost code is particularly expensive to run, as constructing the model for a binary tree is at worst quadratic in the size of the tree, and contracts contain quantifications on the maximal size of the tree that call functions which themselves quantify over the same size in their own contracts or code. In addition, the expensive operation of constructing the model is performed repeatedly in contracts, as SPARK does not yet provide a let-expression form. As a result, inserting one element in a tree of size one takes 2 minutes.

5 Related Work

There have been several previous attempts at verifying red-black trees implementations. In particular, red-black trees are used in the implementation of ordered sets and maps in the standard library of the Coq proof assistant [1,7]. As part of these libraries, the implementations have been proven correct using interactive proofs in Coq. These implementations notably differ from our work because they are written in a functional style, using recursive data types instead of pointers and recursive functions instead of loops. Similar libraries are provided for the Isabelle proof assistant [13]. Functional implementations of red-black trees have also been verified outside of proof assistants, using characteristic formulas [3], or in the Why3 programming language as part of VACID-0 competition [15]. This last implementation differs from the previous ones in that it is mostly auto-active, even if it uses Coq for a few verification conditions.

Verifying imperative implementations of red-black trees is more challenging as it involves reasoning about the well-formedness of the tree structure, which comes for free in the functional implementations. As part of VACID-0, attempts have been made at verifying red-black trees in C using VCC and in Java using KeY [2]. Both attempts seem to have been left in preliminary stages though.

More recently, imperative implementations of red-black trees in C and Java have been verified using more specialized logics. Enea et al. obtained an automatic verification of a C implementation of red-black trees using separation logic, a logic specialized for the verification of heap manipulating programs [6]. In the same way, Stefănescu et al. were able to verify several implementations of red-black trees in particular in Java and C using matching logic [20]. As used in this work, matching logic provides a very precise, low-level view of the heap structure, allowing for powerful proofs on this kind of programs. Both works

use specialized tools, which are specifically designed for verifying low-level, heap manipulating programs but which have never been used, to the best of our knowledge, to verify higher-level software.

6 Conclusion

In this article, we have explained how, using auto-active techniques, we could achieve formal verification of key functional properties of an imperative implementation of red-black trees in SPARK. This is not an example of what should be a regular use of the SPARK toolset but rather a successful demonstration of how far we can go using such technology.

However, the techniques presented on this example can be reused with significant benefits on a much smaller scale. In particular, we have shown that inductive proofs can be achieved rather straightforwardly using auto-active reasoning. The multi-layered approach, using type invariants and model functions to separate concerns, can also be reused to reason about complex data structures.

To popularize the use of auto-active techniques, we are also working on integrating simple interactive proof capabilities in GNATprove. This would allow applying the same techniques in a simpler, more straightforward way, and also to avoid polluting the program space with ghost code which is never meant to be executed.

Acknowledgements. We would like to thank our colleague Ben Brosgol and the anonymous reviewers for their useful comments.

References

1. Appel, A.W.: Efficient verified red-black trees (2011), https://www.cs.princeton.edu/~appel/papers/redblack.pdf
2. Bruns, D.: Specification of red-black trees: Showcasing dynamic frames, model fields and sequences. In: Wolfgang, A., Richard, B. (eds.) 10th KeY Symposium (2011)
3. Charguéraud, A.: Program verification through characteristic formulae. ACM Sigplan Notices 45(9), 321–332 (2010)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition. The MIT Press, 3rd edn. (2009)
5. Dross, C., Moy, Y.: Abstract Software Specifications and Automatic Proof of Refinement, pp. 215–230. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-33951-1_16
6. Enea, C., Sighireanu, M., Wu, Z.: On automated lemma generation for separation logic with inductive definitions. In: International Symposium on Automated Technology for Verification and Analysis. pp. 80–96. Springer (2015)
7. Filliâtre, J.C., Letouzey, P.: Functors for proofs and programs. In: European Symposium on Programming. pp. 370–384. Springer (2004)
8. Filliâtre, J.C., Paskevich, A.: Why3 – Where Programs Meet Provers. In: ESOP'13 22nd European Symposium on Programming. vol. 7792. Springer, Rome, Italy (Mar 2013), https://hal.inria.fr/hal-00789533

9. Furia, C.A., Nordio, M., Polikarpova, N., Tschannen, J.: Autoproof: auto-active functional verification of object-oriented programs. International Journal on Software Tools for Technology Transfer pp. 1–20 (2016), http://dx.doi.org/10.1007/s10009-016-0419-0
10. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S., Zill, B.: Ironfleet: Proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles. pp. 1–17. SOSP '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2815400.2815428
11. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B.: Ironclad apps: End-to-end security via automated full-system verification. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. pp. 165–181. OSDI'14, USENIX Association, Berkeley, CA, USA (2014), http://dl.acm.org/citation.cfm?id=2685048.2685062
12. Kosmatov, N., Marché, C., Moy, Y., Signoles, J.: Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014. In: 7th International Symposium on Leveraging Applications. p. 16. 7th International Symposium on Leveraging Applications, Springer, Corfu, Greece (Oct 2016), https://hal.inria.fr/hal-01344110
13. Lammich, P., Lochbihler, A.: The isabelle collections framework. In: International Conference on Interactive Theorem Proving. pp. 339–354. Springer (2010)
14. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. pp. 348–370. LPAR'10, Springer-Verlag, Berlin, Heidelberg (2010), http://dl.acm.org/citation.cfm?id=1939141.1939161
15. Leino, K.R.M., Moskal, M.: Vacid-0: Verification of ample correctness of invariants of data-structures, edition 0 (2010)
16. Leino, K.R.M., Moskal, M.: Usable auto-active verification. In: Usable Verification Workshop (2010), http://fm.csl.sri.com/UV10/
17. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press (2015)
18. O'Neill, I.: SPARK – a language and tool-set for high-integrity software development. In: Boulanger, J.L. (ed.) Industrial Use of Formal Methods: Formal Verification. Wiley (2012)
19. Polikarpova, N., Tschannen, J., Furia, C.A.: A Fully Verified Container Library, pp. 414–434. Springer International Publishing, Cham (2015), http://dx.doi.org/10.1007/978-3-319-19249-9_26
20. Stefănescu, A., Park, D., Yuwen, S., Li, Y., Roşu, G.: Semantics-based program verifiers for all languages. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 74–91. ACM (2016)
21. Tafat, A., Marché, C.: Binary heaps formally verified in Why3. Research Report 7780, INRIA (Oct 2011), http://hal.inria.fr/inria-00636083/en/